Recuperatorio Parcial 2 – 2019s2 - Estructuras de Datos – UNQ

Nave Espacial

En este examen modelaremos una `Nave` como un tipo abstracto, el cual nos permite construir una nave espacial, dividida en sectores, a los cuales podemos asignar tripulantes y componentes. Para esto, damos por hecho que:

- `data Componente = LanzaTorpedos | Motor Int | Almacen [Barril]`
- `data Barril = Comida | Oxigeno | Torpedo | Combustible`
- El tipo `Sector` es un tipo abstracto, y representa al sector de una nave, el cual contiene componentes y tripulantes asignados.
- El tipo `Tripulante` es un tipo abstracto, y representa a un tripulante dentro de la nave, el cual tiene un nombre, un rango y sectores asignados.
- El tipo `SectorId` es sinónimo de `String`, e identifica al sector de forma unívoca.
- Los tipos `Nombre` y `Rango` son sinónimos de `String`. Todos los nombres de tripulantes son únicos.
- Un sector está vacío cuando no tiene tripulantes, y la nave está vacía si no tiene ningún tripulante.
- Puede haber tripulantes sin sectores asignados.

Representación

Dicho esto, la representación será la siguiente (*que no es posible modificar*):

```
data Nave = N (Map SectorId Sector) (Map Nombre Tripulante) (MaxHeap Tripulante)
```

Esta representación utiliza:

- Un `Map` que relaciona para cada `SectorId` su sector correspondiente.
- Otro `Map` que relaciona para cada `Nombre` de tripulante el tripulante con dicho nombre.
- Una `MaxHeap` que incluye a todos los tripulantes de la nave, cuyo criterio de ordenado es por rango de los tripulantes.

Ejercicios

Invariantes

a) Dar invariantes de representación válidos según la descripción de la estructura.

Implementación

Implementar la siguiente interfaz de `Nave`, utilizando la representación y los costos dados, calculando los costos de cada subtarea, y siendo T la cantidad de tripulantes y S la cantidad de sectores:

b) `construir :: [SectorId] -> Nave`
Propósito: Construye una nave con sectores vacíos, en base a una lista de identificadores de sectores.
Eficiencia: $O(S)$

c) `ingresarT :: Nombre -> Rango -> Nave -> Nave`
Propósito: Incorpora un tripulante a la nave, sin asignarle un sector.
Eficiencia: $O(\log T)$

d) `sectoresAsignados :: Nombre -> Nave -> Set SectorId`
Propósito: Devuelve los sectores asignados a un tripulante.
Precondición: Existe un tripulante con dicho nombre.
Eficiencia: $O(\log M)$

e) `datosDeSector :: SectorId -> Nave -> (Set Nombre, [Componente])`
Propósito: Dado un sector, devuelve los tripulantes y los componentes asignados a ese sector.
Precondición: Existe un sector con dicho id.
Eficiencia: $O(\log S)$

f) `tripulantesN :: Nave -> [Tripulante]`
Propósito: Devuelve la lista de tripulantes ordenada por rango, de mayor a menor.
Eficiencia: $O(\log T)$

g) `agregarASector :: [Componente] -> SectorId -> Nave -> Nave`
Propósito: Asigna una lista de componentes a un sector de la nave.
Eficiencia: $O(C + \log S)$, siendo C la cantidad de componentes dados.

h) `asignarASector :: Nombre -> SectorId -> Nave -> Nave`
Propósito: Asigna un sector a un tripulante.
Nota: No importa si el tripulante ya tiene asignado dicho sector.
Precondición: El tripulante y el sector existen.
Eficiencia: $O(\log S + \log T + T \log T)$

Usuario

Implementar las siguientes funciones **como usuario** del tipo `Nave`, indicando la eficiencia obtenida para cada operación:

i) `sectores :: Nave -> Set SectorId`
Propósito: Devuelve todos los sectores no vacíos (con tripulantes asignados).

j) `sinSectoresAsignados :: Nave ->[Tripulante]`
Propósito: Devuelve los tripulantes que no poseen sectores asignados.

k) `barriles :: Nave -> [Barril]`
Propósito: Devuelve todos los barriles de los sectores asignados de la nave.

Bonus

l) Dar una posible representación para el tipo `Sector`, de manera de que se pueda cumplir con el orden dado para cada operación de la interfaz, pero sin implementarlas.

Anexo de interfaces

`Sector`, siendo C la cantidad de contenedres y T la cantidad de tripulantes:

`crearS :: SectorId -> Sector`	$O(1)$
`sectorId :: Sector -> SectorId`	$O(1)$
`componentesS :: Sector -> [Componente]`	$O(1)$
`tripulantesS :: Sector -> Set Nombre`	$O(1)$
`agregarC :: Componente -> Sector -> Sector`	$O(1)$
`agregarT :: Nombre -> Sector -> Sector`	$O(\log T)$

`Tripulante`, siendo S la cantidad de sectores:

`crearT :: Nombre -> Rango -> Tripulante`	$O(1)$
`asignarS :: SectorId -> Tripulante -> Tripulante`	$O(\log S)$
`sectoresT :: Tripulante -> Set SectorId`	$O(1)$
`nombre :: Tripulante -> String`	$O(1)$
`rango :: Tripulante -> Rango`	$O(1)$

`Set`, siendo N la cantidad de elementos del conjunto:

`emptyS :: Set a`	$O(1)$
`addS :: a -> Set a -> Set a`	$O(\log N)$
`belongsS :: a -> Set a -> Bool`	$O(\log N)$
`unionS :: Set a -> Set a -> Set a`	$O(N \log N)$
`setToList :: Set a -> [a]`	$O(N)$
`sizeS :: Set a -> Int`	$O(1)$

`MaxHeap`, siendo M la cantidad de elementos en la heap:

`emptyH :: MaxHeap a`	$O(1)$
`isEmptyH :: MaxHeap a -> Bool`	$O(1)$
`insertH :: a -> MaxHeap a -> MaxHeap a`	$O(\log M)$
`maxH :: MaxHeap a -> a`	$O(1)$
`deleteMaxH :: MaxHeap a -> MaxHeap a`	$O(\log M)$

`Map`, siendo K la cantidad de claves distintas en el map:

`emptyM :: Map k v`	$O(1)$
`assocM :: k -> v -> Map k v -> Map k v`	$O(\log K)$
`lookupM :: k -> Map k v -> Maybe v`	$O(\log K)$
`deleteM :: k -> Map k v -> Map k v`	$O(\log K)$
`domM :: Map k v -> [k]`	$O(K)$